Linköpings Universitet
Institutionen för datavetenskap (IDA)
UPP-gruppen 2018-03-15

Dubbellänkad lista

Mål

I denna laboration ska du först se ett nytt sätt att få hjälp med *minneshanteringen* i C++. Därefter ska du ta abstraktion till nya nivåer genom att implementera en *iterator* för en given länkad lista och därefter generalisera så valfri datatyp kan användas i listan med hjälp av *C++ mallar*. Som avslutning ska du använda en *namnrymd* för att undvika framtida problem med namngivningskollisioner.

Målet är alltså att göra listan mer generellt användbar; oavsett projekt och behov av datatyp ska listan gå använda på samma sätt som andra standardkomponenter.

Tidsuppskattning och tips

De olika deluppgifterna tar inte lika lång tid att lösa. Vi uppskattar att deluppgift 1 och 2 kan lösas relativt enkelt för den som är väl insatt i implementationen. Senare deluppgifter tar *avsevärt* mycket längre tid. Planera för detta!
Det kan vara frestande att lösa stora delar på en gång, men ju mindre ni ändrar mellan kompileringar desto lättare är det att isolera fel.

Läsanvisningar

- Exempel, Iteratorer [1, kapitel 20.3 - 20.6, s. 720 - 741]
- Exempel, Mallar [1, kapitel 19.3, s. 678 - 693]
- Exempel, Namnrymder [1, kapitel 8.7, s. 294 - 297]
- Referens, `unique_ptr` [1, kapitel B.6.5, s.1167]
- Referens, Iteratorer [1, kapitel B3, s. 1139 - 1143]
- Referens, Mallar [1, kapitel A13, s. 1121 - 1125]
- Referens, Namnrymder [1, kapitel A15, s. 1127]

Referenser

[1] Bjarne Stroustrup, *Programming, Principles and Practices Using C++*, Addison Wesley, 2nd edition, 2014.

Uppgift: Dubbellänkad lista

I denna laboration ska ni utgå ifrån en given implementation av en dubbellänkad lista.

Deluppgift 0: Läs given kod

Läs igenom och se till att ni förstår alla delar av den givna koden. Detta är en grundförutsättning för att kunna lösa övriga uppgifter på ett bra sätt. Det är ert ansvar att be kurspersonal om hjälp för att förstå de delar ni inte själva kan läsa er till.

Instuderingsfrågor

Kan du svara på dessa frågor om den givna implementationen?

- Hur många noder har en tom lista?
- Var läggs ett element när det stoppas in först?
- Var läggs ett element när det stoppas in sist?
- Det finns två at-funktioner, varför?
- Hur fungerar egentligen icke-const versionen av at?

Deluppgift 1: Minneshantering

Utgå ifrån den givna koden. Din första uppgift är att se över minneshanteringen i den givna koden. Just nu saknas destruktorer och vi lagrar vanliga pekare vilket såklart leder till minnesläckor. Detta ska ni åtgärda genom att utnyttja standardkomponenten `std::unique_ptr`. En `unique_ptr` är en typ av smartpekare som har ansvar för en specifik resurs i form av dynamiskt minne. En smartpekare återlämnar automatiskt sin resurs i sin destruktor och löser därmed många återkommande problem man kan få med dynamisk minneshantering. Det som är speciellt med `unique_ptr` är att endast en kan äga en resurs åt gången och automatiskt återlämnar resursen när `unique_ptr`-objektet destrueras. Tänk på att vi inte måste ändra alla pekare till `unique_ptr`, endast de som faktiskt äger en resurs ska vara en smartpekare.

Deluppgift 2: Testprogram

Utgå ifrån resultatet i deluppgift 1 och utöka det givna testprogrammet med bra testfall för att testa de givna funktionerna heltäckande.
TIPS: Programmet `valgrind` är (ganska) bra på att testa bland annat minneshantering. Starta med `valgrind a.out` för ett enkelt test (det finns andra tester man kan göra med valgrind).

Deluppgift 3A: Iterator

Utgå ifrån resultatet i deluppgift 2 och lägg till en egenbyggd iterator (av kategori *bidirectional*) till din lista. Detta görs genom att skapa en ny klass, med namn `List_Iterator` och lägga till medlemmar enligt nedan.

- Nedanstående publika typdeklarationer (namnen är viktiga då de används av standardbiblioteksfunktioner):

  iterator_category Typ av iterator, här `std::bidirectional_iterator_tag` från `<iterator>`.

  value_type Typen som ges vid avreferering.

  difference_type Typ på värdet man får när man subtraherar två iteratorer (exempelvis `int`).

  pointer Pekare till `value_type`.

  reference Referens till `value_type`.
- Stegningsoperatorer för att stega till nästa respektive föregående element, både i prefix- och postfixformat. Detta blir fyra medlemsfunktioner.
- Möjlighet att komma åt värdet i nuvarande nod med avrefereringsoperatorn (`operator*`).
- Jämföra en iterator med en annan (endast likhet och olikhet). Man jämför genom att se om iteratorerna refererar till samma nod i en lista.
- Diverse datamedlemmar och speciella medlemsfunktioner efter behov.

Organisera er iterator så att endast listan har möjlighet att skapa iteratorobjekt. Ni ska även lägga till medlemsfunktionen `begin` i listan för att skapa en iterator som refererar till första elementet och `end` för att generera en förbi-sista iterator.

Observera att endast listan och iteratorn ska behöva känna till nodtypen. Denna får alltså inte förekomma som parameter eller returtyp i publika medlemsfunktioner.

Skapa ett separat testprogram för att testa iteratorn. Om den är korrekt implementerad ska det även fungera att stega baklänges genom er lista med hjälp av reverse-iteratorer:

```
List lst{2,3,1,5};
auto rb { make_reverse_iterator(lst.end()) };
auto re { make_reverse_iterator(lst.begin()) };
for ( auto it = rb; it != re; ++it )
{
  cout << *it << ' ';
}
// ska ge utskriften 5 1 3 2
```

Deluppgift 3B: Mallifiera listan

Utgå ifrån koden från deluppgift 2 (dvs inte med iteratorn) och gör listan mer generell genom att låta användaren ange datatypen på det som ska lagras i listan. För att lösa detta ska ni använda C++ mallar (`template`). Gör listan till en mall och gå igenom koden för att lista ut vilka deklarationer som behöver ändras.
Modifiera även testprogrammet för att använda den nya listan.

För att kompilera en mall behöver kompilatorn se hela definitionen för att se om den angivna typen går att använda. Därför är det lättast att inkludera er implementationsfil längst nere i er deklarationsfil (men inom inkluderingsgarden). Implementationsfilen ska därmed inte längre anges vid kompilering.

Deluppgift 4: Slå ihop del 3A och 3B

Nu ska ni skapa en fullständigt mallifierad lista med iteratorer. Det som återstår nu är alltså att slå ihop föregående uppgifter och mallifiera iteratorn. Eftersom vi kan ha en mer komplex datatyp i listan (exempelvis en string) ska ni även lägga till medlemsåtkomstoperatorn `operator->` till er iterator. Det som är speciellt med denna är att den ska returnera adressen till det lagrade värdet i aktuell listnod.

Modifiera testprogrammet så att den nya funktionaliteten testas.

Deluppgift 5: Namnrymd

Skapa ett nytt tomt testprogram som inkluderar er lista. Skapa dessutom en ny klass i testprogrammet. Den nya klassen ska ha namnet `List`. Klassen kan i övrigt vara helt tom. Skapa även en main-funktion som försöker skapa en lista. Testa kompilera, vad händer?

En lösning på problemet är såklart att inte skapa en ny klass med samma namn som vår förra, men det känns lite jobbigt att inte kunna använda ett bra namn bara för att det redan råkar komma med i en inkluderingsfil. Ni ska åtgärda detta genom att skapa en namnrymd `List_NS` och lägga till er mallifierade listan samt iteratortypen i denna namnrymd.

Modifiera era tidigare testprogram så att de fungerar (kompilerar) med den nya namnrymden. Nu ska det gå att skapa klassen List samtidigt som listan inkluderas och används.